Auto-parallelization of Pidgin C

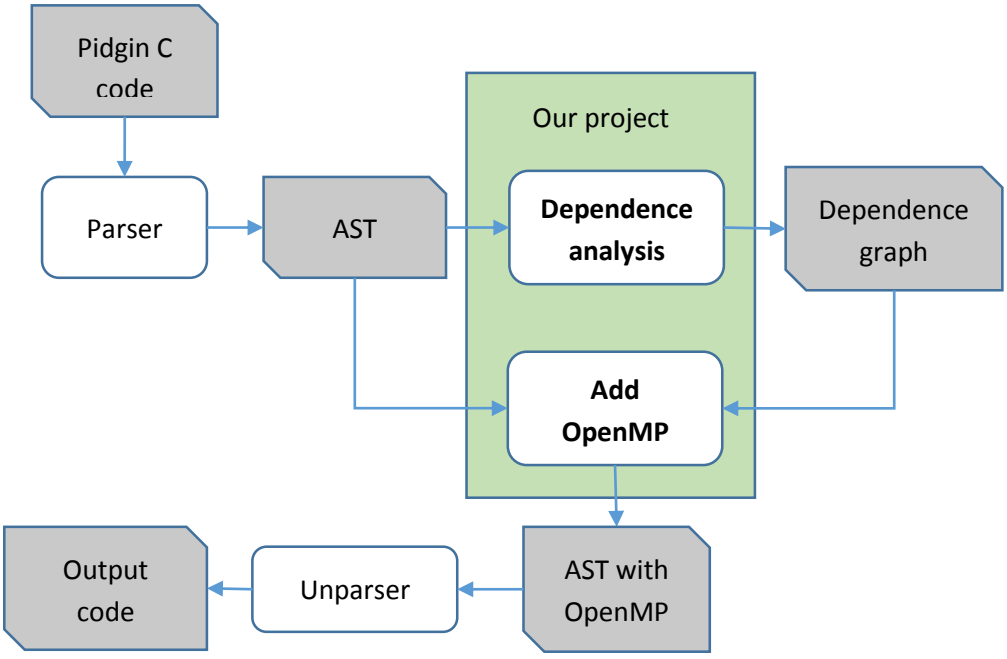
- P523 final project

Tanghong Qiu (taqiu@indiana.edu) Zi Wang (wang417@indiana.edu)

Introduction and Background

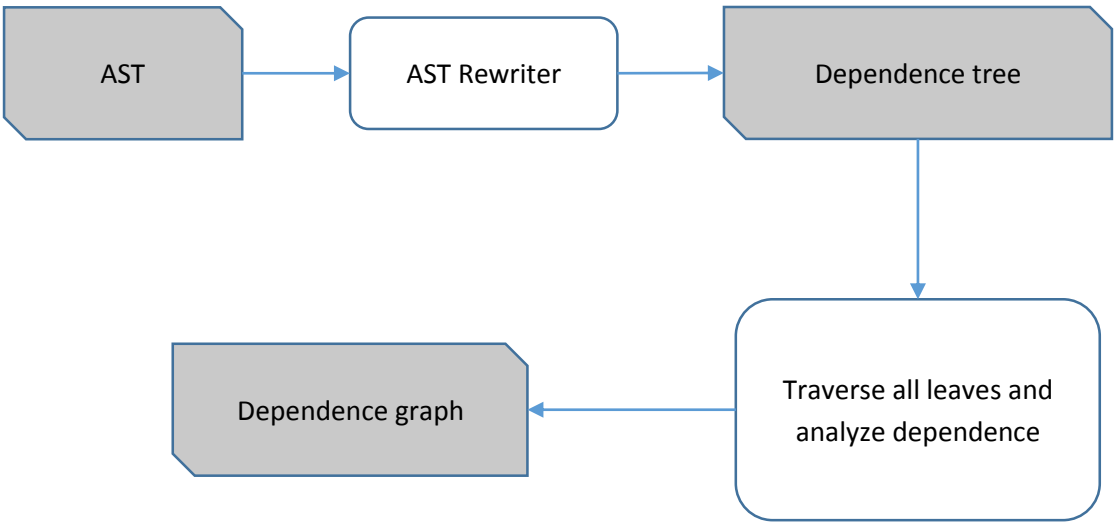
Parallelization is an importance approach to utilize multiple processors and reduce the running time of program. The goal of this project is to implement an auto-parallelization complier for Pidgin C. This project will involve detecting Pidgin C loops that can be parallelized and generating OpenMP directives that allow those loops to be run in parallel. OpenMP is a set of compiler directives that help the application developer to parallelize their workload, and this project mainly utilize the "for directive" which helps share iterations of a loop between a group of threads.

According to the loop parallelization theorem, it is valid to convert a sequential loop to a parallel loop if the loop carries no dependence [1]. Therefore, to detect the parallelizable loop, the compile need to analyze the dependence of target loop, which is the major work of this project. This step will generate a dependence graph which is used to figure out the outermost parallelizable loop and add OpenMP directives.



The project is based on our previous parser assignment, and this project will analyze the abstract syntax tree from Pidgin C parser. This project output a new AST with OpenMP directive according to the dependence graph, and unparse the AST to Pidgin C finally.

Dependence analysis



- **AST Transformation**

The abstract syntax tree is the original intermediate representation for Pidgin C. The dependence graph is able to be derived from the AST directly, but the AST contains a lot of useless information for dependence analysis. Actually, the dependence analysis is concerned only with the variables in expression or statement.

Firstly, all statements should be simplify to a uniform intermediate representation which only keep the variable and its subscript, and we define several classes to save those information. The id field of statement class is a unique label for all statement, and the left and right field will save the variables on the assignment left hand side and right respectively. The variable class will save the identifier and its subscripts.

```
class Statement
    def initialize(id, left, right)
        @id = id
        @left = left
        @right = right
    end
end
```

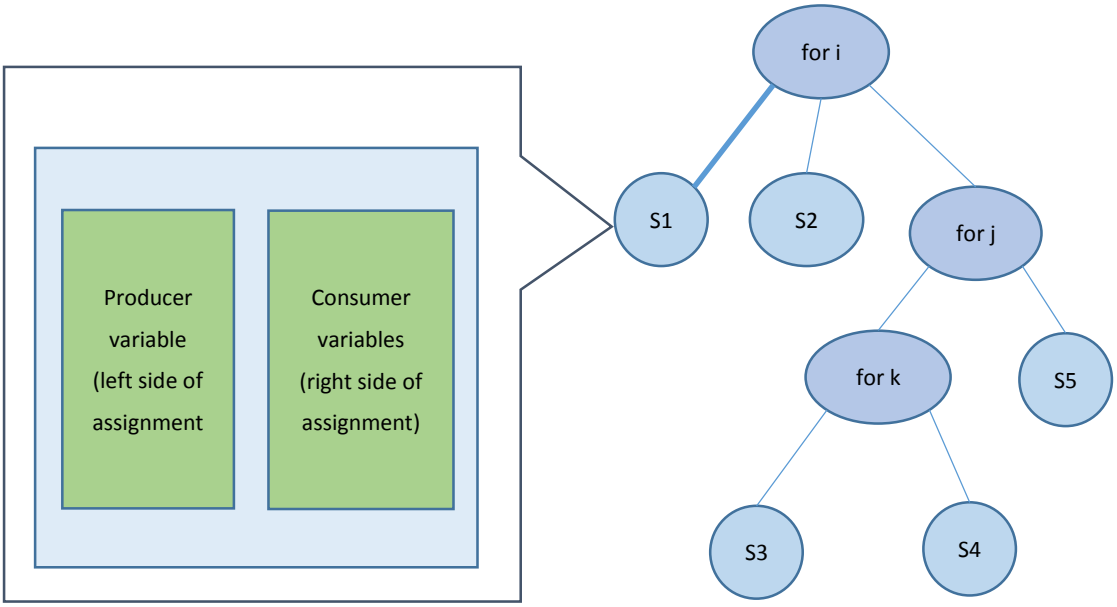
```
class Variable
    def initialize(id, subs)
        @id = id
        @subs = subs
    end
end
```

```
class Subscript
    def initialize(var, offset)
        @var = var
        @offset = offset
    end
end
```

Beside simple statements, the compound statements (e.g. if…else.. , while…) are also allowed in the “for loop”. The statements in the block are addressed as simple statements, and the conditions of compound statements are regarded as statements which only have right side variables.

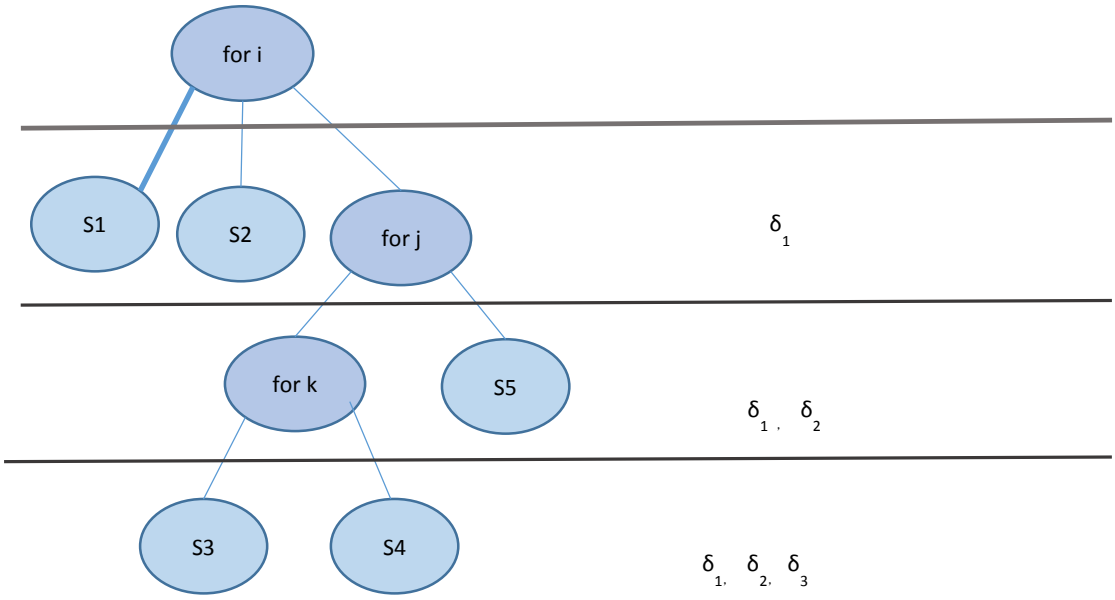
The statements are stored in a tree. The root node is a “for loop” node which contains a list of

statement at its level or nested for loop nodes. Initially, we use a link list to represent the relationship of statement, but the link list cannot represent complicate “for loop” structure. Therefore, we use a tree which can exactly represent the sequence and level of statement.



- **Tree Traversal Algorithm**

After generating the tree for dependence analysis, we need to develop a tree traversal algorithm to take out every two node and analyze their dependence. First of all, we must guarantee the entire tree are traversed and no node is left out. Secondly, we have to reduce the duplicate analysis.



From the above figure, we can see that the different level of loops exits different level dependences. The loop-carried dependence of statements only exist at current or above level. Therefore, we employ the depth first algorithm to traverse every node the tree. Meanwhile, we generate a list of statement nodes at current and deep level, and analyze the dependence between the node and the list. To minimize the time of analysis, the list only contains the nodes with equal or greater id and the nodes at deep levels.

S1	<->	S1, S2, S3, S4, S5	dependence at level [i]
S2	<->	S2, S3, S4, S5	dependence at level [i]
S3	<->	S3, S4	dependence at level [i, j, k]
S4	<->	S4	dependence at level [i, j, k]
S5	<->	S3, S4, S5	dependence at level [i ,j]

- **Dependence between two statements**

- **Dependence Graph**

A general representation for graph is using vertex set and edge set. (G = <V, E>) In this project, we don't care about the nodes which have no dependence with other nodes, so we only need to keep the edge set.

Edge = {statement1, statement2, level}

In the project, the types of dependences are not important for adding Open directive either. However, the level of dependence is required, so we need to save it.

Adding OpenMP directive

After generating the dependence graph, our compile will generate a new AST with OpenMP directive according to the original AST and the dependence graph. The strategy is to find the outermost level loop which have no dependence at this level. And then we will use the Pidgin C unparser to transform the AST back to Pidgin C code.

First, a hash table is used to count the number of dependences at each levels. Secondly, the compile traverses the original AST tree from the outer level to the inner level. If no dependence exists at a "for loop" level, an OpenMP for directive is added to this level. The remained child node won't be traversed if a directive has been added.

Conclusion and further work

All in all, the compile works well with most examples in the text book. However, the compiler only support the general type subscript (e.g. [i-2], [j], [k+3], [9]), and subscripts like [i+k] or [9-j] are not supported in our project.

If a “for loop” node have more than one sub “for loops” at the same level, the compiler can’t generate correct dependence graph for this case. Because we don’t have a feasible approach to represent the dependence graph between the parallelized sub “for loops”. The dependences between the parallelized sub “for loops” are more difficulty to decide, because the indexes of loop can be used repeatedly.

The adding OpenMP directive process can also be improved further. Initially, we decided to develop an algorithm from the codegen algorithm, but we realized the vectorization is different from the parallelization. The parallelization requires no dependence at the parallelized level. One of the removing dependence approach is to separate the loop, but this method will make the program create and join the threads repeatedly. It might reduce the performance of program. Another advanced transformation is to change the position of for loop header, for example, the header of no-dependence for loop can be moved to outermost level and be parallelized. This approach can improve the performance of parallelization.

As we discussed above, this project can be extended further in different aspects. But the time for project is limited, so we just use the most reliable approach to do the auto-parallelization.

Reference

[1] Allen & Kennedy, Optimizing Compilers for Modern Architectures, 1st Edition